HUST

ĐẠI HỌC BÁCH KHOA HÀ NỘI

HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

EDGE AI SUMMER SCHOOL 2026 · SoICT — HUST

Thị giác máy tính hiệu năng cao trên thiết bị biên Góc nhìn từ các mô hình lightweight

Tổng quan nghiên cứu 2024–2026 · 10–14/8/2026 · Hà Nội

Thị giác máy tính hiệu năng cao trên thiết bị biên: Góc nhìn từ các mô hình lightweight

High-Performance Computer Vision on Edge Devices: A Lightweight Model Perspective

Tổng quan các nghiên cứu mới nhất 2024–2026

Trường hè Edge AI 2026 · ĐHBK Hà Nội · 10–14 tháng 8, 2026 Đồng tài trợ: Tập đoàn SAMSUNG × Tập đoàn NAVER

Đối tượng và mục tiêu của bài giảng

ĐỐI TƯỢNG PHÙ HỢP

- Đã nắm CNN và ViT ở mức cơ bản
- Cập nhật kết quả SOTA 2024–2026
- Muốn hiểu vì sao mô hình nhỏ vẫn đạt độ chính xác cao
- Tìm hiểu hướng nghiên cứu về Edge AI

MỤC TIÊU ĐẦU RA

- Tổng quan các backbone nhẹ SOTA kèm số liệu
- Bốn kỹ thuật nén: lượng tử hóa, chưng cất, cắt tỉa, NAS
- Hiểu YOLO26, EfficientSAM3 và lý do thiết kế
- Nguyên lý thiết kế

5 phần

0 **Vì sao Edge AI?**
Động lực · ứng dụng · tam giác đánh đổi · giới hạn của FLOPs · phần cứng

1 **Backbone siêu nhẹ SOTA**
MobileNetV4 · iFormer · RepViT · SHViT · MicroViT · TinyNeXt · Mamba

2 **Nén mô hình**
Quantization INT8/INT4/NVFP4 · distillation · pruning · NAS

3 **Từ mô hình đến tác vụ**
YOLO v11 → v12 → 26 · thu nhỏ SAM · chuỗi công cụ · kịch bản ứng dụng

4 **Huấn luyện tại biên & hướng nghiên cứu**
Federated Learning · xu hướng 2026+ · tổng kết

HUST

hust.edu.vn fb.com/dhbkhn

0

ĐỘNG LỰC & KHUNG TƯ DUY

Vì sao phải đưa AI xuống biên?

Động lực · ứng dụng · tam giác đánh đổi · giới hạn của chỉ số FLOPs · phần cứng.

Vai trò Cloud AI và Edge AI

CLOUD AI

Mô hình khổng lồ, hàng tỉ tham số, GPU/TPU datacenter

Round-trip mạng → không đáp ứng thời gian thực

Dữ liệu phải rời thiết bị: rủi ro riêng tư + chi phí băng thông

EDGE AI

Suy luận tại chỗ, hoạt động cả khi mất mạng

Giảm dữ liệu thô rời thiết bị; riêng tư vẫn cần cơ chế bảo vệ

Ràng buộc cứng: RAM vài trăm MB, chạy pin, tản nhiệt thụ động

Edge không thay thế cloud: cloud phù hợp với huấn luyện và mô hình lớn; edge phù hợp với suy luận cục bộ, độ trễ thấp và giảm truyền dữ liệu thô.

[1] H. B. McMahan et al., *"Communication-Efficient Learning of Deep Networks from Decentralized Data,"* AISTATS, 2017.
[2] L. Zhu, Z. Liu, and S. Han, *"Deep Leakage from Gradients,"* NeurIPS, 2019.

Động lực đẩy AI xuống biên

Độ trễ

Xe tự hành ở 60 km/h đi 1,7 m mỗi 100 ms. Phản xạ phanh phải dưới 100 ms → cloud không kịp.

Riêng tư

Ảnh y tế, camera nhà máy, khuôn mặt người dùng: pháp lý và đạo đức đều đòi dữ liệu ở lại thiết bị.

Năng lượng

Truyền dữ liệu qua mạng thường tốn thêm năng lượng so với xử lý cục bộ. Suy luận local tiết kiệm pin hơn gửi ảnh đi.

Kết nối

Drone nông nghiệp, camera vùng sâu: mạng chập chờn hoặc không có. Mô hình phải hoạt động độc lập.

mô hình tốt = tốt trong RÀNG BUỘC, không phải tốt tuyệt đối trên các bảng xếp hạng

Sáu nhóm ứng dụng tiêu biểu

Xe tự hành

Nhiều camera, lidar, radar đồng thời tạo dữ lieu liên tục, cần ra quyết định xử lý tại xe với độ trễ thấp

Smart Factory

Kiểm tra lỗi sản phẩm theo thời gian thực ngay trên dây chuyền; edge computer đặt gần camera và máy sản xuất.

Thiết bị đeo y tế

Theo dõi ECG, nhịp tim và tín hiệu sinh lý; xử lý cục bộ giúp phản hồi nhanh và giảm truyền dữ liệu nhạy cảm.

Nông nghiệp chính xác

Drone và robot nhận diện cây trồng, sâu bệnh tại chỗ; hữu ích khi năng lượng và kết nối mạng bị giới hạn.

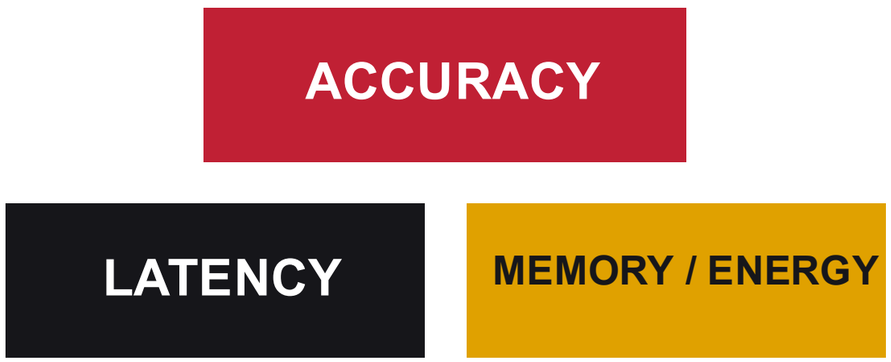
Thành phố thông minh

Phân tích nhiều luồng camera gần nguồn dữ liệu; giảm băng thông truyền video và hỗ trợ cảnh báo thời gian thực.

Điện thoại / AR

Phân vùng ảnh, nhận diện và hiệu ứng camera thời gian thực; AI thường được tăng tốc bằng NPU/Neural Engine trên thiết bị

Đánh đổi đa mục tiêu: accuracy - latency - resources



Các mục tiêu thường xung đột; thiết kế tốt tìm điểm cân bằng phù hợp với ngân sách triển khai

Đặt đúng câu hỏi

Không phải “mô hình nào tốt nhất”, mà hỏi“ với latency/ memory này, mô hình nào cho accuracy cao nhất?”.

Tạo ra SOTA =

Cùng latency → accuracy cao hơn; hoặc cùng accuracy → latency, memory hay energy thấp hơn.

Mỗi nền tảng có điểm tối ưu khác nhau

MCU, NPU điện thoại, Jetson: cùng bài toán nhưng ba lựa chọn mô hình khác nhau → cùng bài toán nhưng có thể có các mô hình khác nhau

FLOPs không hoàn toàn là tốc độ: ít phép tính chưa chắc chạy nhanh

VÌ SAO FLOPS KHÔNG PHẢN ÁNH TỐC ĐỘ

Latency không chỉ phụ thuộc số phép tính mà còn phụ thuộc **truy cập bộ nhớ, loại toán tử, mức độ song song và khả năng tối ưu của phần cứng/runtime**. Vì vậy, hai mô hình có FLOPs tương đương vẫn có thể chạy rất khác nhau trên chip thật.

Đo trên phần cứng nào?

iPhone ANE, Jetson Orin, EdgeTPU, CPU ARM … cùng một model có thể cho latency rất khác nhau

Đo latency thế nào?

End-to-end hay chỉ inference? Có gồm tiền/hậu xử lý không? Batch size bao nhiêu? Báo p50 hay p95 (độ trễ điển hình hay độ trễ ở trường hợp chậm)?

Kết quả?

Nên báo cáo nhiều model/cấu hình ở các mức accuracy - latency - memory/energy khác nhau, thay vì chỉ chọn một model “tốt nhất”.

Compute-bound hay memory-bound: nhìn qua arithmetic intensity

ARITHMETIC INTENSITY

AI = số phép tính /số bytes dữ liệu di chuyển

AI thấp thường nằm ở vùng memory-bound: tăng năng lực tính toán chưa chắc giúp nhanh hơn nếu nút thắt vẫn là băng thông bộ nhớ.

- AI thấp + dưới ngưỡng của phần cứng → memory-bound
  AI cao + vượt ngưỡng → có xu hướng compute-bound

CÁC TRƯỜNG HỢP DỄ BỊ MEMORY-BOUND

Depthwise conv: Mỗi kênh được xử lý gần như độc lập nên số phép tính khá ít. Nhưng dữ liệu vẫn phải được đọc từ bộ nhớ và ghi kết quả trở lại. Vì ít tính toán nhưng vẫn phải di chuyển nhiều dữ liệu, tốc độ dễ bị giới hạn bởi băng thông bộ nhớ.

Phân giải cao: Feature map càng lớn thì càng chiếm nhiều bộ nhớ. Khi không còn nằm vừa trong cache, chip phải thường xuyên lấy dữ liệu từ RAM/DRAM: vốn chậm và tốn năng lượng hơn nhiều so với truy cập cache.

Nhiều nhánh hoặc nhiều toán tử nhỏ: Mỗi nhánh có thể tạo ra tensor trung gian riêng. Sau đó hệ thống phải ghép, tách, đọc và ghi các tensor này nhiều lần. Dù tổng FLOPs không cao, chi phí di chuyển dữ liệu và gọi nhiều kernel vẫn có thể làm chương trình chậm.

Attention chưa fuse: Attention thường có nhiều bước liên tiếp như tạo ma trận điểm số, softmax rồi nhân tiếp. Nếu mỗi bước đều ghi kết quả trung gian ra bộ nhớ rồi đọc lại ở bước sau, lượng dữ liệu di chuyển rất lớn. Fused attention cố gắng gộp nhiều bước lại để giữ dữ liệu gần compute unit hơn và giảm đọc/ghi bộ nhớ.

Memory-bound xảy ra khi chip dành nhiều thời gian "chờ dữ liệu được mang tới" hơn là thực sự tính toán.

Nhiều kỹ thuật tối ưu Edge AI cùng hướng tới một mục tiêu: giảm lượng dữ liệu trung gian, tận dụng lại dữ liệu tốt hơn và hạn chế việc đọc/ghi bộ nhớ.

Phần cứng Edge AI: mô hình phải phù hợp với thiết bị đích

NỀN TẢNG	TÀI NGUYÊN	MÔ HÌNH PHÙ HỢP	ỨNG DỤNG
MCU / TinyML	Hàng chục KB → vài MB RAM; điện năng rất thấp	Mô hình cực nhỏ ví dụ MCUNet	Cảm biến thông minh, IoT
Điện thoại / NPU	GB RAM, giới hạn pin & nhiệt	Lightweight CNN/ Hybrid (iFormer, MobileNetV4, RepViT)	Camera, AR/XR, xử lý ảnh on-device
Jetson / Edge GPU	4–64 GB, ~7–60 W tùy thiết bị	Detection, segmentation (YOLO26, EfficientViT-SAM)	Robot, xe tự hành, camera công nghiệp
Edge server	GPU rời, tài nguyên và công suất lớn hơn	Nhiều model đồng thời, batch nhỏ, multi-stream	Gateway nhà máy, nhiều camera, hạ tầng cục bộ

Không chọn model chỉ theo accuracy: phải kiểm tra peak memory, latency, operator support và power trên chính phần cứng đích.

HUST

hust.edu.vn fb.com/dhbkhn

1

BACKBONE SIÊU NHẸ · SOTA 2024–2026

Các hướng thiết kế backbone hiệu quả

Các mô hình mới và nguyên lý thiết kế rút ra từ chúng

Các làn sóng kiến trúc:

~2012–2020

CNN

MobileNetV1–V3, EfficientNet.
Giỏi bắt đặc trưng cục bộ, chạy hiệu quả trên phần cứng; thông tin toàn cục được tích lũy dần qua nhiều lớp.

2020 →

ViT

ViT, DeiT, Swin.
Cơ chế chú ý giúp mô hình nắm bắt quan hệ giữa các vùng ở xa nhau; đổi lại cần nhiều tính toán và bộ nhớ hơn.

2022 →

Hardware-aware / Efficient

MobileViT, EfficientViT, FastViT, RepViT, MobileNetV4, iFormer, SHViT.
Kết hợp thế mạnh của tích chập và cơ chế chú ý, ưu tiên độ chính xác, độ trễ và bộ nhớ khi chạy trên thiết bị thực.

2024 →

SSM/ Vision Mamba

VMamba, EfficientViM.
Nắm bắt thông tin toàn cục với chi phí tăng tuyến tính theo kích thước đầu vào; một hướng thay thế cho cơ chế chú ý.

Các hướng mới không xóa hướng cũ: chúng kế thừa và tái kết hợp những thiết kế hiệu quả.

Chi phí của các khối cơ bản: tính toán và bộ nhớ

KHỐI	CHI PHÍ TÍNH TOÁN	HÀNH VI TRÊN BỘ NHỚ
Tích chập 3×3 thông thường	$O(H \cdot W \cdot C^2 \cdot 9)$	Một vùng dữ liệu đầu vào được tái sử dụng nhiều lần cho nhiều phép nhân–cộng. Vì vậy mỗi lần đọc dữ liệu có thể "làm được nhiều việc", tận dụng cache và phần cứng khá tốt.
Depthwise 3x3 + pointwise 1×1	$O(H \cdot W \cdot C \cdot 9 + H \cdot W \cdot C^2)$	Depthwise xử lý từng kênh riêng, nên mỗi phần tử được đọc vào nhưng chỉ tham gia ít phép tính. Dữ liệu vẫn phải đọc/ghi gần như đầy đủ → số FLOPs giảm mạnh nhưng lưu lượng bộ nhớ không giảm tương ứng.
Self-attention	$O(N^2 \cdot d)$	Phải tạo và xử lý nhiều tensor trung gian, đặc biệt là ma trận quan hệ giữa các token. Nếu các tensor này được ghi ra bộ nhớ rồi đọc lại ở bước sau, lượng dữ liệu di chuyển tăng rất nhanh khi số token lớn.
Selective scan (SSM)	Tuyến tính theo N	Không cần lưu ma trận quan hệ N×N như attention. Trạng thái được cập nhật tuần tự qua các token, nên lượng bộ nhớ trung gian tăng chậm hơn; tuy nhiên tốc độ thực tế vẫn phụ thuộc việc kernel có giữ được dữ liệu gần bộ xử lý hay phải đọc/ghi bộ nhớ nhiều lần.

N = số phần tử/token trong chuỗi. Các công thức chỉ cho biết xu hướng chi phí; tốc độ thực tế vẫn phải đo trên phần cứng và trình thực thi mục tiêu.

MobileNetV4: kiến trúc hiệu quả trên nhiều phần cứng di động

Ý TƯỞNG & KIẾN TRÚC

Mục tiêu “universal”

Dòng MobileNetV4 được thiết kế để đạt hiệu quả tốt trên nhiều loại phần cứng: CPU, GPU, DSP…

UIB - Universal Inverted Bottleneck

Một khối linh hoạt với các depthwise convolution có thể bật/tắt; từ cùng một cấu trúc có thể tạo ra nhiều kiểu block như Inverted Bottleneck, ConvNeXt, FFN và ExtraDW.

Mobile MQA

Nhiều đầu truy vấn dùng chung key/value → giảm dữ liệu phải đọc/ghi; nhanh hơn **hơn 39%** so với multi-head attention trên các bộ tăng tốc di động trong thử nghiệm của paper.

NAS hai giai đoạn

Tìm cấu trúc tổng thể trước, sau đó tinh chỉnh chi tiết → tăng hiệu quả tìm kiếm và mở rộng không gian model.

KẾT QUẢ

87%

Top-1 ImageNet-1K, MNv4-Hybrid-L sau distillation

3,8 ms

latency trên Pixel 8 EdgeTPU

Phần lớn đạt điểm cân bằng rất tốt giữa độ chính xác và độ trễ trên nhiều nền tảng được benchmark

UIB: hai depthwise tùy chọn tạo bốn biến thể

CẤU HÌNH TỔNG QUÁT

`[DW?] → 1×1 expand → [DW?] → 1×1 project`

UIB thêm hai vị trí depthwise tùy chọn vào cấu trúc inverted bottleneck. NAS quyết định có dùng từng depthwise hay không và có thể chọn kích thước kernel.

Bài học thiết kế: tạo một khối đủ linh hoạt, rồi để NAS chọn cấu hình phù hợp với độ chính xác và phần cứng mục tiêu.

CẤU HÌNH	DW TRƯỚC MỞ RỘNG	DW SAU MỞ RỘNG	Ý NGHĨA
FFN	×	×	Chỉ còn hai lớp 1×1 để biến đổi và trộn thông tin giữa các kênh
ConvNeXt-like	✓	×	Trộn thông tin không gian trước khi mở rộng số kênh → rẻ hơn
Inverted Bottleneck	×	✓	Mở rộng kênh trước rồi mới depthwise → khả năng biểu diễn cao hơn nhưng tốn hơn
ExtraDW	✓	✓	Dùng cả hai depthwise → tăng độ sâu và vùng quan sát

Mobile MQA: dùng chung key/value để giảm băng thông bộ nhớ

MULTI-HEAD (MHA)

Mỗi head có Q, K, V riêng. Vì vậy phải tạo và xử lý nhiều bộ K/V, làm tăng lượng dữ liệu cần đọc/ghi bộ nhớ.

Tốc độ thực tế còn phụ thuộc cách triển khai; trên phần cứng, việc đọc/ghi dữ liệu từ bộ nhớ có thể trở thành nút thắt.

MULTI-QUERY (MQA)

Vẫn giữ nhiều query head, nhưng tất cả dùng chung một head K và một head V → giảm lượng K/V cần tính, lưu và đọc lại.

Mobile MQA giúp phần attention nhanh hơn 39% so với MHA trên các accelerator di động được thử nghiệm, với chênh lệch accuracy chỉ −0,03 điểm %.

Điểm cần nhớ: MQA không chỉ giảm phép tính mà quan trọng hơn là giảm lượng K/V phải di chuyển qua bộ nhớ: đặc biệt có lợi khi băng thông là nút thắt.

iFormer: kết hợp ConvNet × Transformer, tối ưu theo độ trễ trên iPhone

Ý TƯỞNG & KIẾN TRÚC

Các tầng đầu: ưu tiên tích chập

Feature map ở các tầng đầu còn lớn nên dễ bị giới hạn bởi truy cập bộ nhớ. iFormer giữ các phép tích chập đã được tối ưu tốt trên phần cứng, đồng thời thay LayerNorm bằng BatchNorm và phân bổ lại số block giữa các stage để giảm độ trễ

Stage 3 và 4: SHMA khi feature map nhỏ

Khi độ phân giải nhỏ hơn, đưa attention vào để học quan hệ toàn cục mà không làm chi phí tăng quá mạnh.

Single-Head Modulation Attention

hai nhánh: một nhánh giữ đặc trưng ảnh, một nhánh dùng attention để xác định ngữ cảnh quan trọng. Hai nhánh được nhân với nhau để attention điều chỉnh đặc trưng cần giữ hay giảm.

Thiết kế theo latency đo thật

Mỗi thay đổi được kiểm tra trực tiếp trên iPhone 13, thay vì chỉ nhìn FLOPs.

KẾT QUẢ

80,4%

top-1 tại 1,10 ms trên iPhone 13

~1,4×

nhanh hơn FastViT-SA12 ở cùng accuracy tương đương

Vượt MobileNetV4-Conv-M khoảng 0,5 điểm ở độ trễ tương đương.

SHMA: dùng single-head attention để tạo ngữ cảnh điều khiển đặc trưng

CƠ CHẾ

Ngữ cảnh: Đặc trưng → Self-Attention một head → thông tin toàn cục

Đặc trưng → phép chiếu tuyến tính

Đầu ra = sigmoid(đặc trưng) × sigmoid(ngữ cảnh)

Một nhánh giữ và biến đổi đặc trưng hiện tại; một nhánh dùng attention để xác định thông tin toàn cục quan trọng. Hai nhánh được nhân với nhau để ngữ cảnh điều chỉnh đặc trưng đầu ra.

Vẫn có softmax attention

- Các token vẫn được so sánh với nhau để xác định mức độ liên quan.
- Softmax biến các điểm liên quan thành trọng số chú ý, từ đó quyết định token nào cần được lấy thông tin nhiều hơn.

→ SHMA không loại bỏ attention; nó chỉ giảm từ nhiều head xuống một head.

Giảm chi phí của Multi-head

- Dùng một head giúp giảm các thao tác chia–ghép và sắp xếp lại dữ liệu của multi-head attention.
- Q và K cũng được giảm số chiều, giúp giảm thêm lượng tính toán và dữ liệu phải xử lý.

→ Mục tiêu là giữ khả năng nhìn toàn cục nhưng làm attention nhẹ hơn trên thiết bị di động.

Lợi ích được đo

- Nhanh hơn: MHA → SHA giảm độ trễ từ 1,40 ms xuống 1,12 ms trên iPhone 13, tương đương khoảng 1,25× nhanh hơn.
- Độ chính xác giảm rất ít: chuyển sang một head chỉ làm giảm khoảng 0,1 điểm Top-1.
- SHMA bù lại chất lượng: SHA lấy thông tin toàn cục; SHMA dùng thông tin đó để điều chỉnh đặc trưng, giúp bù phần độ chính xác bị mất mà vẫn giữ lợi thế về độ trễ.

Paper: iFormer: Integrating ConvNet and Transformer for Mobile Application (ICLR 2025)

Đặt attention ở đâu? Phụ thuộc số token

TẦNG NÔNG

56×56 = 3 136 token

Nếu dùng attention toàn cục, phải xét khoảng: $3.136^2 \approx 9{,}8$ triệu cặp token
→ Chi phí tính toán và bộ nhớ rất lớn.

→ Conv, giảm độ phân giải

TẦNG GIỮA

28×28 = 784 token

Số cặp giảm còn: $784^2 \approx 615$ nghìn
→ Nhẹ hơn khoảng 16× so với tầng 56×56; có thể bắt đầu dùng attention nhẹ.

→ Conv + attention nhẹ

TẦNG SÂU

7×7 = 49 token

Chỉ còn:
$49^2 = 2.401$ cặp token
→ Attention toàn cục lúc này rẻ hơn nhiều; đặc trưng cũng thường mang thông tin ngữ nghĩa cao hơn.

→ Attention / SSM

Từ 3.136 xuống 49 token, số cặp token giảm 4.096×. Vì vậy nhiều kiến trúc hybrid xử lý cục bộ ở tầng nông và đưa cơ chế tương tác toàn cục vào các tầng sâu hơn.

RepViT: CNN thuần học lại các lựa chọn thiết kế của ViT

Ý TƯỞNG & KIẾN TRÚC

Hướng tiếp cận ngược lại

Bắt đầu từ MobileNetV3, rồi lần lượt đưa các nguyên tắc thiết kế hiệu quả của ViT nhẹ vào CNN.

Tách xử lý không gian và xử lý kênh thành hai phần riêng:

Depthwise conv trộn thông tin giữa các vị trí; conv 1×1 trộn thông tin giữa các kênh

Tái tham số hóa cấu trúc

Huấn luyện với nhánh phụ để học tốt hơn; khi suy luận gộp lại để giảm chi phí tính toán và bộ nhớ.

CNN thuần, không self-attention

Giữ toàn bộ backbone bằng convolution, thuận lợi cho triển khai trên thiết bị di động

KẾT QUẢ

80,0%

top-1 ImageNet, RepViT-M1.0

1,0 ms

trên iPhone 12; RepViT-M1.5 đạt 82,3% ở 1,5 ms (300 epoch + distillation)

Tên M0.9 / M1.0 / M1.5… chính là mức latency mục tiêu đo trên iPhone 12 trong paper.

RepViT: nhiều nhánh khi huấn luyện, một phép tích chập khi suy luận

KHI HUẤN LUYỆN

Nhánh 1: Depthwise Conv 3×3 + BN
Nhánh 2: Depthwise Conv 1×1
Nhánh 3: giữ nguyên đầu vào (identity)
→ cộng ba kết quả → BN

Khi huấn luyện, nhiều nhánh giúp khối có cấu trúc phong phú hơn để học. Nhưng nếu giữ nguyên ba nhánh khi triển khai, thiết bị phải chạy nhiều phép toán rồi cộng kết quả.

KHI SUY LUẬN

Gộp cả ba nhánh thành một Depthwise Conv 3×3 duy nhất rồi BN

- 1×1 → 3×3: thêm số 0 xung quanh kernel 1×1.
- Identity → 3×3: biểu diễn bằng kernel có 1 ở chính giữa, 0 ở các vị trí còn lại.

Kết quả: lúc huấn luyện dùng cấu trúc nhiều nhánh; lúc suy luận chỉ còn một depthwise 3×3 tương đương về mặt toán học → bỏ chi phí của các nhánh và phép cộng trung gian.

SHViT và MicroViT: giảm dư thừa bằng attention một head

SHVIT · CVPR 2024

Dư thừa kép

ở đầu mạng có quá nhiều token nên attention rất đắt; ở các tầng sau, nhiều attention head có thể học thông tin tương tự nhau.

Giảm token sớm + attention một head

SHViT giảm độ phân giải sớm bằng convolution để giảm số token. Ở các tầng sau, chỉ một phần đặc trưng đi qua attention một head; phần còn lại được giữ lại rồi ghép với kết quả.

Nhanh trên cả GPU lẫn mobile

SHViT-S4 nhanh hơn MobileViTv2 ×1.0 khoảng 3,3× trên GPU, 8,1× trên CPU và 2,4× trên iPhone 12, đồng thời cao hơn 1,3 điểm Top-1 trong benchmark của paper

MICROVIT · ISCAS 2025

ESHA: Efficient Single-Head Attention

MicroViT không đưa toàn bộ đặc trưng qua attention. Nó chỉ chọn một phần số kênh để xử lý bằng attention một head; phần còn lại đi theo đường nhẹ hơn rồi được ghép lại.

Giảm số token trước khi tính attention

Key và Value có thể được giảm độ phân giải, nên attention phải so sánh ít token hơn → giảm tính toán và đọc/ghi bộ nhớ.

Kết quả

Thiết kế hướng tới cả độ trễ thấp và tiết kiệm năng lượng, thay vì chỉ giảm FLOPs.
Paper báo cáo tốc độ cải thiện tới khoảng 3,6× và hiệu quả năng lượng tăng hơn 40% trong các thiết lập so sánh của họ.

Điểm chung: với mô hình nhỏ, nhiều head có thể dư thừa. Giảm số head, số token và số kênh đi qua attention có thể cải thiện tốc độ và năng lượng mà vẫn giữ độ chính xác cạnh tranh.

Papers: SHViT: Single-Head Vision Transformer with Memory Efficient Macro Design (CVPR 2024) · MicroViT: A Vision Transformer with Low Complexity Self Attention for Edge Device (ISCAS 2025)

Vì sao single-head có thể giữ accuracy trong ViT nhẹ?

1 Nhiều head có thể học thông tin giống nhau

Trong các ViT nhỏ mà SHViT khảo sát, nhiều head ở các tầng sau tạo attention map khá giống nhau. Với DeiT-T, độ tương đồng trung bình giữa các head ở tầng sau lên tới **78,3%**; việc bỏ một số head cũng chỉ làm accuracy thay đổi ít.

→ **Ý nghĩa:** không phải head nào cũng đóng góp thông tin hoàn toàn khác biệt.

2 Multi-head tạo thêm chi phí đọc/ghi bộ nhớ

Chia attention thành nhiều head cần thêm các thao tác sắp xếp lại dữ liệu như reshape. Paper cho thấy một phần đáng kể thời gian của Multi-Head Self-Attention có thể nằm ở các thao tác thiên về truy cập bộ nhớ như reshape và normalization, chứ không chỉ ở phép nhân ma trận

→ **Ý nghĩa:** nhiều head hơn không chỉ thêm khả năng biểu diễn mà còn có thể thêm overhead trên phần cứng thật.

3 Không cần cho toàn bộ đặc trưng đi qua attention

SHViT chỉ chọn một phần số kênh để đưa qua single-head attention; các kênh còn lại được giữ nguyên rồi ghép lại. Cuối khối có phép chiếu để thông tin từ attention lan sang toàn bộ số kênh.

→ **Ý nghĩa:** dùng attention đúng phần cần thiết thay vì xử lý mọi kênh bằng attention.

Trong các ViT nhẹ được khảo sát, nhiều attention head có thể dư thừa. SHViT vì vậy dùng một head trên một phần số kênh để giữ khả năng mô hình hóa toàn cục nhưng giảm chi phí tính toán và truy cập bộ nhớ.

Paper: SHViT: Single-Head Vision Transformer with Memory Efficient Macro Design (CVPR 2024)

TinyNeXt: đưa backbone ImageNet xuống mức 1M tham số

Ý TƯỞNG & Ý NGHĨA

Mục tiêu: khoảng 1M tham số

rất nhỏ so với nhiều backbone mobile truyền thống.

Vượt các mốc tham chiếu

TinyNeXt-T: 1,0M tham số, 71,5% Top-1 ImageNet-1K - cao hơn MobileNetV1 (70,6%, 4,2M) và PVTv2-B0 (70,5%, 3,7M)

1M weights ≈ 4 MB FP32 hoặc ≈ 1 MB INT8

Đây chỉ là trọng số; triển khai thực còn phụ thuộc activation RAM, workspace và runtime.

Mở rộng không gian TinyML vision

Thu hẹp đáng kể kích thước backbone và hướng tới thiết bị biên tài nguyên hạn chế. Paper xác nhận hiệu quả trên Jetson Nano và ARM Cortex-A57.

SỐ LIỆU

1,0M

tham số - quy mô rất nhỏ cho backbone ImageNet

71,5%

top-1 ImageNet-1K

TinyNeXt-T cho thấy backbone chỉ khoảng 1M tham số vẫn có thể đạt độ chính xác cạnh tranh trên ImageNet-1K. Khả năng triển khai thực tế vẫn phải kiểm tra RAM, latency và phần cứng đích.

Không có mô hình tối ưu

MÔ HÌNH	THAM SỐ	TOP-1	PHÙ HỢP
TinyNeXt-T	1,0M	**71,5%**	Cần backbone cực nhỏ; hướng tới TinyML/edge hạn chế tài nguyên, nhưng phải kiểm tra RAM thực tế trên thiết bị đích
SHViT-S4	~16,5M	**79,4%**	CPU/GPU/mobile; ưu tiên throughput và giảm overhead của attention
RepViT-M1.5	14,0M	**82,3%**	Điện thoại; CNN thuần, ~1,5 ms trên iPhone 12
iFormer-M	~9M	**80,4%**	Điện thoại; ưu tiên latency rất thấp: 1,10 ms trên iPhone 13
MNv4-Hybrid-L	~37M	**87,0%**	Phù hợp với thiết bị có bộ tăng tốc AI mạnh; đạt 3,8 ms trên Pixel 8 EdgeTPU.

Một số kết quả sử dụng distillation hoặc recipe huấn luyện khác nhau. Các số liệu chỉ dùng để định vị quy mô và mục tiêu triển khai, không phải xếp hạng trực tiếp giữa các mô hình.

Papers: An Efficient Hybrid Vision Transformer for TinyML Applications · SHViT: Single-Head Vision Transformer with Memory Efficient Macro Design
RepViT: Revisiting Mobile CNN From ViT Perspective · iFormer: Integrating ConvNet and Transformer for Mobile Application
MobileNetV4: Universal Models for the Mobile Ecosystem

State-Space Models: tránh việc so sánh mọi cặp token

NÚT THẮT CỦA ATTENTION

Mọi token phải tương tác với nhau: Self-attention toàn cục tạo quan hệ giữa mọi cặp token, nên khi số token tăng, chi phí tăng rất nhanh.

Tốn cả tính toán và bộ nhớ: Ma trận attention lớn làm tăng lượng dữ liệu trung gian và đọc/ghi bộ nhớ.

→ Chi phí tang N^2

HƯỚNG GIẢI CỦA MAMBA / SSM

Không tạo quan hệ giữa mọi cặp token: SSM đi qua các token và duy trì một trạng thái gọn để truyền thông tin.

Selective scan: học cách nhớ và quên: Mô hình quyết định thông tin nào cần giữ lại khi xử lý từng token.

→ Chi phí tăng gần tuyến tính theo N

Attention kết nối trực tiếp mọi token nhưng chi phí tăng nhanh; SSM truyền thông tin qua trạng thái nên phù hợp hơn khi số token lớn.

Selective scan: học cách giữ và bỏ thông tin

CẬP NHẬT TRẠNG THÁI THEO TỪNG TOKEN

```
Token mới + trạng thái trước
→ quyết định giữ gì, quên gì
→ trạng thái mới
```

Mamba không cập nhật trạng thái theo một quy tắc cố định cho mọi token. Một số tham số thay đổi theo đầu vào, nhờ đó mô hình có thể giữ thông tin quan trọng và bỏ thông tin ít liên quan

VỚI ẢNH 2D: CẦN CHỌN CÁCH QUÉT

Ảnh không có thứ tự tự nhiên như một câu văn. VMamba vì vậy quét ảnh theo bốn hướng để thu thập thông tin từ nhiều phía

Lợi thế

Không cần tạo quan hệ giữa mọi cặp token như self-attention. Số token tăng → chi phí chính tăng tuyến tính, phù hợp hơn khi chuỗi đặc trưng dài.

Điểm cần lưu ý

O(N) không tự động có nghĩa chạy nhanh hơn trên mọi chip. Tốc độ thực còn phụ thuộc cách quét, số hướng quét và cách kernel được triển khai trên phần cứng.

Selective scan có thể hiểu như một bộ nhớ có cổng: đọc từng token, giữ thông tin cần thiết và bỏ phần không quan trọng.

EfficientViM: làm Vision Mamba nhẹ và nhanh hơn

Ý TƯỞNG & KIẾN TRÚC

HSM-SSD: xử lý trong trạng thái ẩn nhỏ hơn

Thay vì thực hiện phần trộn kênh tốn kém trên toàn bộ chuỗi token, EfficientViM thực hiện nó trên trạng thái ẩn đã nén→ ít phép tính hơn và giảm lượng dữ liệu phải xử lý.

Giảm nút thắt bộ nhớ

Thiết kế hạn chế các phép toán phải đọc/ghi nhiều dữ liệu, vì những thao tác này có thể làm mô hình chậm dù FLOPs không cao

Kết hợp trạng thái từ nhiều tầng

Thông tin trạng thái ẩn của nhiều stage (tầng) được kết hợp để bù lại lượng thông tin có thể mất do nén, giúp mô hình nhỏ nhưng vẫn giữ khả năng biểu diễn tốt.

Kết quả

O(N)

Chi phí tăng tuyến tính theo số token

Cân bằng tốc độ – độ chính xác tốt

Trên ImageNet-1K, EfficientViM đạt trade-off tốc độ–độ chính xác tốt hơn các mô hình nhẹ được so sánh; paper báo cáo có cấu hình vừa nhanh hơn SHViT vừa cao hơn tới 0,7 điểm Top-1.

Ý tưởng chính: nén thông tin vào trạng thái ẩn nhỏ, thực hiện phần xử lý tốn kém ở đó, rồi kết hợp thông tin giữa các tầng để giữ độ chính xác.

Năm nguyên lý thiết kế cần ghi nhớ

1 **Phối hợp kiến trúc có chủ đích**

Tầng đầu còn nhiều token → ưu tiên tích chập và giảm độ phân giải; tầng sâu ít token hơn → mới đưa attention hoặc SSM vào.

2 **Dùng attention có chọn lọc**

Giảm số head, số kênh hoặc lượng K/V phải xử lý: SHViT, Mobile MQA, SHMA… đều nhằm giữ lợi ích của attention nhưng giảm chi phí.

3 **Huấn luyện phức tạp, suy luận đơn giản**

Có thể dùng re-parameterization hoặc distillation khi huấn luyện, nhưng mô hình triển khai cuối cùng vẫn cần gọn và dễ chạy.

4 **Thiết kế theo phần cứng đích**

Không chỉ nhìn FLOPs; phải đo latency, bộ nhớ và năng lượng trực tiếp trên chip và runtime mục tiêu.

5 **Thiết kế tổng thể quyết định phần lớn chi phí**

Cách chia stage, thời điểm giảm độ phân giải và phân bố số kênh thường ảnh hưởng lớn hơn các tinh chỉnh nhỏ bên trong từng block.

HUST

hust.edu.vn fb.com/dhbkhn

2

NÉN MÔ HÌNH · SOTA

Khi kiến trúc tốt vẫn chưa đủ nhẹ

Quantization INT8 → INT4 → NVFP4 · distillation · pruning · NAS

Bốn kỹ thuật nén: dùng khi nào

KỸ THUẬT	CƠ CHẾ	KHI NÀO DÙNG	LỢI ÍCH ĐIỂN HÌNH
Lượng tử hóa	Giảm độ chính xác số, ví dụ **FP32 → INT8/INT4**, cho trọng số và có thể cả activation	Thường thử **INT8 hậu huấn luyện** trước; dùng khi phần cứng có hỗ trợ tốt kiểu dữ liệu đó	Giảm kích thước model và băng thông bộ nhớ; FP32→INT8 giảm dung lượng trọng số khoảng **4×**
Chưng cất tri thức	Model nhỏ học từ đầu ra hoặc đặc trưng của model lớn	Khi model nhỏ chưa đạt accuracy mong muốn	Tăng accuracy của model nhỏ **mà không cần teacher khi suy luận**
Cắt tỉa	Loại bỏ kênh, filter, head hoặc trọng số ít quan trọng	Khi kiến trúc đã có nhưng còn dư thừa	Giảm tham số/FLOPs; **tốc độ chỉ tăng rõ khi phần cứng/runtime khai thác được cấu trúc đã cắt**
Tìm kiếm kiến trúc tự động	Tự động thử nhiều kiến trúc và chọn theo accuracy + latency/tài nguyên	Khi phần cứng đích đã xác định và có đủ chi phí tìm kiếm	Tìm kiến trúc phù hợp riêng với **phần cứng và yêu cầu triển khai**

Lượng tử hóa: đưa số thực về các mức số nguyên

LƯỢNG TỬ HÓA AFFINE

```
q = clip(round(x/s) + z, qmin, qmax);  x_new = s·(q − z)

x: số thực ban đầu
q: số nguyên sau lượng tử hóa
x_new: số thực khôi phục gần đúng
s: độ lớn mỗi bước lượng tử
z: vị trí của số 0 trong miền số nguyên
```

Ý tưởng: số thực → làm tròn về một mức số nguyên → khi cần có thể khôi phục lại gần đúng.

Per-tensor

Một scale cho cả tensor: Đơn giản, dễ triển khai; nhưng nếu các kênh có miền giá trị rất khác nhau thì độ chính xác có thể giảm.

Per-channel

Mỗi kênh một scale: Phù hợp hơn khi các kênh có phân bố khác nhau; thường cho độ chính xác tốt hơn khi lượng tử hóa trọng số.

Per-block

Scale theo khối nhỏ: Linh hoạt hơn khi dùng số bit thấp, nhưng cần lưu nhiều scale hơn và phần cứng/runtime phải hỗ trợ.

Chia scale càng chi tiết thì lượng tử hóa càng chính xác, nhưng triển khai cũng phức tạp hơn.

PTQ hay QAT: chọn phương pháp nào

PTQ - POST-TRAINING

Không huấn luyện lại; Mô hình FP32 đã huấn luyện xong mới được chuyển sang INT8/INT4.

Chi phí thấp. Với PTQ tĩnh, thường dùng một tập dữ liệu nhỏ để calibration nhằm chọn scale và zero-point phù hợp.

Với INT8, nên thử PTQ trước; Nếu độ chính xác giảm quá nhiều, mới cân nhắc QAT.

QAT - QUANTIZATION-AWARE

Mô phỏng lượng tử hóa ngay khi huấn luyện.
các giá trị được làm tròn như khi chạy INT8/INT4, giúp mô hình học cách thích nghi với sai số.

Mô hình học cách chịu sai số. Trọng số được điều chỉnh để sau khi chuyển sang INT8/INT4 vẫn giữ độ chính xác tốt hơn.

Tốn thêm thời gian huấn luyện, nhưng hữu ích khi PTQ chưa đạt yêu cầu hoặc khi số bit rất thấp.

Quy trình thực tế: thử PTQ trước → đo độ chính xác và độ trễ → chỉ dùng QAT khi PTQ chưa đạt mục tiêu.

QAT vẫn học được dù có phép làm tròn

VẤN ĐỀ
Phép làm tròn tạo ra các mức rời rạc:
0,1 → 0 0,2 → 0 0,4 → 0
Đầu vào thay đổi nhưng kết quả vẫn giữ nguyên.
→ **Gradient bằng 0 ở hầu hết các điểm**
→ Mô hình khó biết phải cập nhật trọng số theo hướng nào.

STRAIGHT-THROUGH ESTIMATOR
Forward: vẫn làm tròn

→ mô phỏng sai số khi chạy INT8/INT4: mô hình thấy đúng sai số lượng tử hóa

Backward:khi cập nhật trọng số, tạm bỏ qua ảnh hưởng của phép làm tròn để mô hình vẫn học được.
→ cho gradient truyền thẳng qua để trọng số vẫn cập nhật được: chỉ mô hình hướng cần điều chỉnh

Mô hình vừa:
- thấy được sai số lượng tử hóa;
- vừa tiếp tục học được.

Forward: có làm tròn.
Backward: cho gradient đi thẳng.

STE giúp QAT mô phỏng lượng tử hóa khi tính đầu ra nhưng vẫn giữ được gradient để huấn luyện.

Giảm số bit: model nhỏ hơn, nhưng chưa chắc chạy nhanh hơn

FP32
1× weight
Mốc so sánh; độ chính xác số cao nhưng tốn bộ nhớ nhất.

FP16
2× weight
Giảm một nửa dung lượng trọng số. Tốc độ chỉ tăng khi phần cứng có hỗ trợ FP16 hiệu quả.

INT8
4× weight
Thường là lựa chọn thực tế để bắt đầu lượng tử hóa; cần kiểm tra độ chính xác và kernel trên phần cứng đích. TensorRT hỗ trợ INT8 cho cả trọng số và activation.

INT4
8× weight
Nhỏ hơn nữa nhưng sai số lượng tử hóa khó kiểm soát hơn → có thể cần PTQ nâng cao hoặc QAT. Khả năng tăng tốc phụ thuộc mạnh vào backend;

NVFP4
8× weight
Dùng **scale theo từng khối nhỏ** để giữ độ chính xác tốt hơn ở 4 bit. NVFP4 được NVIDIA thiết kế và tăng tốc cho kiến trúc Blackwell

Ít bit → trọng số nhỏ hơn; nhưng latency thực tế còn phụ thuộc kernel, kiểu lượng tử hóa và phần cứng đích.

Vì sao ViT khó lượng tử hóa? ACTIVATION CÓ OUTLIER

VẤN ĐỀ

Một vài giá trị bất thường

Trong ViT, activation ở một số vị trí/lớp có thể lớn hơn nhiều so với phần còn lại

Outlier làm bước lượng tử hóa lớn hơn

Scale phải bao phủ cả giá trị lớn → khoảng cách giữa các mức lượng tử tăng.

Các giá trị nhỏ bị biểu diễn kém chính xác

Nhiều giá trị bình thường phải dùng chung hoặc rất ít mức lượng tử → sai số tăng, đặc biệt ở 4 bit

GIẢI PHÁP CHÍNH

Muốn lượng tử hóa tốt hơn, cần giảm ảnh hưởng của outlier trước khi chuyển activation xuống số bit thấp.
Có thể đi theo các hướng như:

- Điều chỉnh phân bố activation
- Dùng scale chi tiết hơn

Một cách tiếp cận nổi bật là SmoothQuant: chuyển một phần “độ khó” từ activation sang weight trước khi lượng tử hóa.

Không phải mọi giá trị trong ViT đều khó lượng tử hóa như nhau; cần chú ý đặc biệt đến activation có outlier.

SmoothQuant: activation dễ lượng tử hóa hơn

VẤN ĐỀ

Một số kênh activation có giá trị quá lớn (outlier) → scale phải rộng → lượng tử hóa INT8 kém chính xác.

SMOOTHQUANT LÀM GÌ

Activation ÷ d → giá trị nhỏ và đều hơn
Weight × d → bù lại phần đã giảm ⇒ **đầu ra không đổi**

$$20 \times 3 = 60 \cdot 5 \times 12 = 60$$

Lợi ích

Activation bớt outlier → scale nhỏ hơn → dễ lượng tử hóa INT8 hơn.

Có cần huấn luyện lại?

Không. Dùng một tập dữ liệu nhỏ để chọn hệ số d, điều chỉnh weight một lần trước khi triển khai.

Mục tiêu

Lượng tử hóa cả weight và activation xuống INT8 (W8A8) mà hạn chế giảm độ chính xác.

Giảm độ lớn của activation, chuyển sang weight: đầu ra không đổi nhưng activation dễ lượng tử hóa hơn.

Chưng cất tri thức: model nhỏ học từ model lớn

TEACHER → STUDENT

Teacher: mô hình lớn, độ chính xác cao. Student: mô hình nhỏ, cần triển khai nhanh. Khi huấn luyện, student học thêm thông tin từ teacher; khi suy luận chỉ cần student → không tăng chi phí triển khai.

Học đầu ra

Teacher không chỉ nói “đây là mèo” mà cho 90% mèo · 7% linh miêu · 3% chó. Student học mức độ giống nhau giữa các lớp, thay vì chỉ nhãn đúng/sai.

Học đặc trưng

Student tạo ra đặc trưng trung gian giống teacher → hữu ích khi muốn model nhỏ học cách biểu diễn ảnh của model lớn.

Học mối quan hệ

Student học cách teacher nhận biết mẫu nào giống nhau, mẫu nào khác nhau: không cần sao chép chính xác từng đặc trưng.

VÌ SAO HIỆU QUẢ?

Nhãn cứng chỉ nói “đáp án đúng là mèo”. Teacher còn cho biết: “rất giống mèo, hơi giống linh miêu, gần như không giống ô tô”: phần thông tin bổ sung này thường gọi là **dark knowledge**.

Distillation giúp model nhỏ học không chỉ “đáp án”, mà còn học cách model lớn phân biệt các lớp.

Cắt tỉa: giảm tham số chưa chắc giảm độ trễ

UNSTRUCTURED: XÓA TỪNG TRỌNG SỐ

Một số trọng số về 0, nhưng kích thước tensor giữ nguyên: ma trận 100×100 vẫn là 100×100, chỉ nhiều số 0 hơn.

Ưu điểm: giảm lượng trọng số cần lưu (với định dạng sparse phù hợp). Hạn chế: muốn chạy nhanh cần phần cứng và kernel hỗ trợ sparse.

Nhiều số 0 ≠ tự động chạy nhanh hơn.

STRUCTURED: XÓA CẢ NHÓM

Cắt cả kênh, filter hoặc attention head → tensor thật sự nhỏ hơn → số phép tính giảm → dễ tạo tăng tốc thực tế hơn.

Ví dụ: 64 kênh → cắt còn 48 kênh. Phần cứng vẫn xử lý tensor dense bình thường, chỉ là tensor nhỏ hơn.

QUY TRÌNH THƯỜNG DÙNG

Huấn luyện → đánh giá phần ít quan trọng → cắt → fine-tune → đo lại

Fine-tune giúp mô hình phục hồi phần độ chính xác bị mất sau khi cắt.

Unstructured tạo nhiều số 0; structured làm mạng thật sự nhỏ hơn. Tốc độ tăng lên cuối cùng vẫn phải đo trên phần cứng đích.

NAS và kết hợp kỹ thuật nén: phải đo sau từng bước

NAS - TỰ ĐỘNG TÌM KIẾN TRÚC

Thử nhiều cấu hình, chọn kiến trúc phù hợp với **độ chính xác + độ trễ + giới hạn tài nguyên** . MobileNetV4 dùng NAS hai giai đoạn: tìm cấu trúc tổng thể trước, rồi tinh chỉnh chi tiết.

Tìm model phù hợp phần cứng đích ngay từ khâu thiết kế.

CÓ THỂ KẾT HỢP NHIỀU KỸ THUẬT

Distillation → model nhỏ học tốt hơn. Structured pruning → giảm kênh/block. Quantization → giảm số bit weight/activation.

Ví dụ CQKD: kết hợp cả ba trong cùng một framework.

KHÔNG CỘNG/NHÂN SPEEDUP MỘT CÁCH MÁY MÓC

INT8 giúp model nhỏ hơn; pruning giúp giảm phép tính. Nhưng kết hợp hai kỹ thuật không có nghĩa tốc độ sẽ tăng theo phép nhân.

Phải đo lại: độ chính xác · độ trễ · RAM · năng lượng: trên phần cứng đích.

QUY TRÌNH THAM KHẢO

Chọn kiến trúc → distillation/pruning nếu cần → quantization → benchmark sau từng bước.

HUST

hust.edu.vn fb.com/dhbkhn

3

TỪ MÔ HÌNH ĐẾN TÁC VỤ · SOTA

Backbone chỉ là khởi đầu

Detection với YOLO v11 → v12 → 26 · thu nhỏ SAM · chuỗi công cụ · kịch bản ứng dụng.

Phát hiện vật thể thời gian thực: ba mốc tham chiếu gần đây

VÌ SAO YOLO PHỔ BIẾN TRÊN THIẾT BỊ BIÊN

- YOLO dự đoán vị trí và lớp vật thể trong **một mạng duy nhất**, phù hợp với yêu cầu xử lý thời gian thực.
- Nhưng tốc độ thực tế vẫn phụ thuộc kích thước ảnh, hậu xử lý, runtime và phần cứng đích.

2024

YOLO11

Dòng Ultralytics ổn định, dễ triển khai
Có 5 kích cỡ: n, s, m, l, x
Hỗ trợ train, inference và export
Phù hợp làm baseline thực tế cho nhiều bài toán edge.

2/2025

YOLOv12

Đưa attention vào YOLO nhưng vẫn hướng tới thời gian thực
Kiến trúc attention-centric
Dùng Area Attention và R-ELAN
Mục tiêu: tăng khả năng mô hình hóa mà vẫn giữ tốc độ cạnh tranh.
Đây là công trình độc lập, không phải thế hệ tiếp theo chính thức của Ultralytics.

1/2026

YOLO26

Tập trung đơn giản hóa pipeline triển khai
NMS-free mặc định
Bỏ DFL trong regression head
Hướng tới inference end-to-end và triển khai edge đơn giản hơn

Xu hướng gần đây: không chỉ tăng độ chính xác, mà còn giảm chi phí attention và đơn giản hóa toàn bộ pipeline suy luận.

Papers: YOLOv11: An Overview of the Key Architectural Enhancements (paper phân tích, 2024) · YOLOv12: Attention-Centric Real-Time Object Detectors
Ultralytics YOLO26: Unified Real-Time End-to-End Vision Models (2026)

YOLO11: lựa chọn thực dụng để triển khai

KIẾN TRÚC & HỆ SINH THÁI

C3k2 + C2PSA

C3k2 giúp trích đặc trưng hiệu quả; C2PSA thêm attention ở cuối backbone để làm nổi bật thông tin quan trọng.

Năm cỡ: n / s / m / l / x

2,6M đến 56,9M tham số: phủ mọi phần cứng

Hỗ trợ nhiều bài toán thị giác

Detection, segmentation, pose, classification… đều có model YOLO11 tương ứng

Thuận tiện khi triển khai

Ultralytics hỗ trợ export sang nhiều định dạng như ONNX, TensorRT và CoreML bằng cùng API/CLI.

SỐ LIỆU

39,5

mAP@50-95 trên COCO, bản v11n

2,6M

tham số bản nano

.

Paper tham khảo: YOLOv11: An Overview of the Key Architectural Enhancements (2024) · Nguồn kiến trúc chính thức: Ultralytics Docs

YOLOv12: attention tiến vào YOLO

KIẾN TRÚC & KẾT QUẢ

Area-Attention (A²): chia nhỏ vùng tính attention

Chia bản đồ đặc trưng thành vài vùng lớn theo chiều ngang hoặc dọc, rồi tính attention trong từng vùng.
Ít cặp token phải so sánh hơn → giảm chi phí attention nhưng vẫn nhìn được vùng rộng

R-ELAN: giúp mạng học ổn định hơn

Residual ELAN: Thêm đường truyền tắt và tổ chức lại cách ghép đặc trưng

FlashAttention: giảm đọc ghi bộ nhớ trên GPU

Không thay đổi ý nghĩa của attention, nhưng tổ chức phép tính hiệu quả hơn để giảm truy cập bộ nhớ.

SỐ LIỆU

40,6

mAP tại 1,64 ms trên T4 (bản N)

Hơn v10-N 2,1 điểm và v11-N 1,2 điểm ở cùng tốc độ GPU.

LƯU Ý

YOLOv12 cho thấy attention vẫn có thể dùng cho detection thời gian thực nếu giảm số vùng tương tác và tối ưu truy cập bộ nhớ.

Paper: YOLOv12: Attention-Centric Real-Time Object Detectors (2025)

YOLO26: bỏ NMS để đơn giản hóa triển khai

BỐN ĐỔI MỚI CHÍNH

End-to-end, không cần NMS mặc định

YOLO26 dùng **one-to-one head** để tạo dự đoán cuối trực tiếp.
Không cần bước NMS phía sau → pipeline đơn giản hơn, giảm hậu xử lý.

Bỏ DFL

Không còn Distribution Focal Loss trong phần hồi quy bounding box.
Detection head nhẹ hơn và thuận tiện hơn khi chuyển model sang ONNX/TensorRT và các runtime khác

Progressive Loss + STAL

lúc đầu cho nhiều dự đoán cùng học một vật thể; về cuối tập trung vào một dự đoán chính để phù hợp với NMS-free.
STAL: giúp vật thể nhỏ không bị bỏ quên khi gán nhãn.

MuSGD: kết hợp SGD và ý tưởng từ Muon

Muon cân chỉnh hướng cập nhật trọng số để tránh một vài hướng chi phối quá mạnh; YOLO26 kết hợp cách này với SGD để cải thiện quá trình huấn luyện

BENCHMARK CÔNG BỐ

+43%

CPU ONNX: YOLO26n so với YOLO11n trên Intel Xeon 2,0 GHz

40,9–57,5

mAP COCO; 1,7–11,8 ms trên T4 TensorRT, tùy cỡ

Paper: Ultralytics YOLO26: Unified Real-Time End-to-End Vision Models (2026)

NMS-free thay đổi pipeline triển khai như thế nào?

PIPELINE CÓ NMS

Model có thể tạo nhiều hộp giới hạn cho cùng một vật thể.

NMS là hậu xử lý riêng; có thể chạy CPU, GPU hoặc plugin tùy backend: giữ hộp giới hạn tốt nhất trong số các hộp giới hạn trùng nhau

NMS là một bước hậu xử lý riêng, nên làm pipeline triển khai phức tạp hơn.

NMS-FREE (YOLO26)

YOLO26 được huấn luyện để mỗi vật thể chỉ có một dự đoán chính, không cần NMS

Ở chế độ end-to-end mặc định, YOLO26 trả về tối đa 300 detection mỗi ảnh, mỗi detection đã gồm tọa độ hộp giới hạn, độ tin cậy và lớp.

Ít bước hậu xử lý hơn→ code triển khai đơn giản hơn.
độ ổn định latency vẫn cần xác nhận trên backend đích.

Không phải mọi backend đều hỗ trợ NMS-free của YOLO26; một số định dạng như Edge TPU và QNN hiện tự quay về head one-to-many và vẫn cần NMS

Các hướng làm SAM nhẹ hơn - từ SAM1 đến SAM3

Bài toán: image encoder ViT-H của SAM gốc khoảng **632M tham số:** là thành phần nặng nhất của mô hình.

1 - MobileSAM / EdgeSAM

Thay phần xử lý ảnh lớn bằng mạng nhỏ hơn
MobileSAM dùng TinyViT, EdgeSAM dùng RepViT-M1.

Mạng nhỏ học lại từ SAM lớn để giữ khả năng phân vùng nhưng giảm chi phí tính toán.

2 - EfficientSAM

Dạy mạng nhỏ học đặc trưng của SAM lớn
Trong lúc huấn luyện, che một phần ảnh và yêu cầu mạng nhỏ học cách tái tạo đặc trưng của SAM gốc.

Không chỉ thay mạng nhỏ hơn, mà còn tìm cách huấn luyện nó tốt hơn.

3 - EfficientViT-SAM

Giữ cách dùng SAM, chỉ thay phần xử lý ảnh
Giữ phần nhận prompt và tạo mask, thay ViT-H bằng EfficientViT.

Mục tiêu: giữ khả năng của SAM nhưng chạy nhanh và nhẹ hơn.

4 - EfficientSAM3 (preprint 2025)

Nén thêm phần xử lý video
Với video, mô hình còn phải nhớ thông tin từ các khung hình trước.
EfficientSAM3 làm nhẹ cả:
phần xử lý ảnh → phần ghi nhớ video → toàn bộ mô hình

Mục tiêu: giảm chi phí không chỉ cho ảnh đơn mà cả khi xử lý video.

Giữ khả năng prompt của SAM: thay hoặc chưng cất các thành phần nặng để giảm chi phí triển khai.

Papers: Segment Anything · Faster Segment Anything: Towards Lightweight SAM for Mobile Applications
EdgeSAM: Prompt-In-the-Loop Distillation for On-Device Deployment of SAM · EfficientSAM: Leveraged Masked Image Pretraining for Efficient Segment Anything
EfficientViT-SAM: Accelerated Segment Anything Model Without Performance Loss
EfficientSAM3: Progressive Hierarchical Distillation for Video Concept Segmentation from SAM1, 2, and 3 (preprint)

Cách làm mô hình thị giác lớn trở nên nhỏ hơn

1

Giữ lại phần cần thiết

Không nhất thiết thay toàn bộ mô hình
→ Giữ những phần vẫn hoạt động tốt, chỉ tối ưu phần nặng nhất. Ví dụ SAM: giữ phần nhận prompt hay tạo mask

2

Thay phần nặng bằng phần nhẹ

Phần xử lý ảnh thường chiếm nhiều chi phí
Thay encoder lớn bằng encoder nhỏ hơn, ví dụ ViT-H/L → RepViT, TinyViT, EfficientViT
→ Giảm tham số. Bộ nhớ, và thời gian xử lý

3

Dùng mô hình lớn để dạy mô hình nhỏ

Mô hình nhỏ học đặc trưng hoặc đầu ra từ mô hình lớn.
Sau đó huấn luyện thêm trên bài toán cần triển khai..
→ Mục tiêu: model nhỏ hơn nhưng vẫn giữ được phần lớn khả năng của model lớn.

Công thức chung: giữ phần cần thiết → thay phần nặng → dùng mô hình lớn để dạy mô hình nhỏ.

Papers: EfficientViT-SAM: Accelerated Segment Anything Model Without Accuracy Loss
EfficientSAM3: Progressive Hierarchical Distillation for Video Concept Segmentation from SAM1, 2, and 3 (preprint)

Từ mô hình huấn luyện đến chạy trên thiết bị

1

Huấn luyện

Huấn luyện mô hình bằng PyTorch/JAX và lưu trọng số.

2

Chuyển định dạng

Chuyển mô hình sang định dạng phù hợp để triển khai, ví dụ ONNX.

3

Tối ưu cho phần cứng

Dùng TensorRT, TFLite, CoreML, OpenVINO… tùy thiết bị.

4

Đo trên thiết bị thật

Không chỉ đo thời gian chạy của model mà cần:
độ trễ · RAM · năng lượng · nhiệt độ

BA LƯU Ý KHI ĐO

① Chọn mô hình theo phần cứng triển khai, không theo accuracy ② Đo sau khi thiết bị đã chạy ổn định, không lấy lần chạy đầu tiên ③ Tốc độ mô hình nhanh chưa chắc hệ thống nhanh nếu Tiền/hậu xử lý còn chậm

Xe tự hành / robot giao hàng — chọn mô hình theo độ trễ thực tế

RÀNG BUỘC

Xử lý nhanh mỗi khung hình

Camera 30 FPS → một ảnh mỗi ~33 ms. Mục tiêu 15–30 ms/model là ngân sách tham khảo — kiểm tra trên thiết bị thật.

Độ trễ ảnh hưởng phản ứng

Ở 60 km/h: chậm thêm 30 ms → xe đi thêm ~0,5 m. Không chỉ nhìn độ trễ trung bình : theo dõi cả p95/p99.

Chạy được trên phần cứng đích

Jetson Orin dùng được TensorRT, nhưng vẫn kiểm tra: model có được hỗ trợ · FP16/INT8 có nhanh hơn · RAM có đủ.

HƯỚNG LỰA CHỌN

1 - Bắt đầu với detector nhỏ

YOLO26-s/m là ứng viên để benchmark, không mặc định tốt nhất. Đo trực tiếp accuracy và latency trên Jetson Orin.

2 - Kiểm tra vật thể nhỏ

Người đi bộ, biển báo ở xa chỉ chiếm ít pixel - đánh giá riêng trên dữ liệu thực tế.

3 - Thử FP16 trước, INT8 sau

FP16 dễ triển khai hơn; chỉ dùng INT8 nếu giữ accuracy và thực sự giảm latency.

4 - Đo toàn bộ pipeline

Camera → tiền xử lý → model → hậu xử lý → kết quả. Không chỉ đo riêng phần mạng.

Model tốt nhất không phải model mAP cao nhất: mà là model đạt đủ độ chính xác trong ngân sách độ trễ của phần cứng đích.

Smart Factory: kiểm tra lỗi ngay trên dây chuyền

RÀNG BUỘC

Dây chuyền chạy liên tục

Sản phẩm đi qua camera liên tục - mô hình phải xử lý kịp tốc độ của dây chuyền.

Lỗi có thể nhỏ và thay đổi

Vết xước, nứt, thiếu chi tiết… có thể rất nhỏ - cần phát hiện tốt cả những vùng lỗi nhỏ.

Ảnh sản phẩm cần được bảo mật

Dữ liệu có thể cần xử lý ngay trong nhà máy - không cần gửi ảnh lên cloud.

HƯỚNG LỰA CHỌN

1 - Phát hiện lỗi: detector nhỏ

Thử YOLO11n / YOLO26n rồi đo trực tiếp trên Jetson hoặc PC công nghiệp. Chọn model theo độ chính xác + tốc độ thực tế.

2 - Khoanh chính xác vùng lỗi: segmentation

Có thể dùng EfficientViT-SAM để tách chính xác vùng được chỉ định. SAM không tự biết đâu là lỗi - nó tách vùng khi đã có điểm, hộp hoặc vùng cần quan tâm.

3 - Tối ưu ngay trên thiết bị

Thử FP16, sau đó INT8 nếu cần. Đo lại độ chính xác · độ trễ · RAM sau khi tối ưu.

Mô hình tốt là mô hình vừa đủ chính xác, chạy kịp dây chuyền và giữ dữ liệu tại chỗ.

HUST

hust.edu.vn fb.com/dhbkhn

4

TƯƠNG LAI · KẾT NỐI TRƯỜNG HÈ

Huấn luyện ngay trên biên

Federated Learning · sáu hướng nghiên cứu 2026+ · tổng kết bốn điều mang về.

Học liên kết: dữ liệu ở lại thiết bị, nhưng chưa chắc đã riêng tư

FEDAVG HOẠT ĐỘNG NHƯ THẾ NÀO?

① Server gửi mô hình chung xuống các thiết bị → ② mỗi thiết bị học bằng dữ liệu của riêng mình → ③ gửi **cập nhật mô hình** , không gửi dữ liệu gốc → ④ server gộp thành mô hình mới → lặp lại.

Dữ liệu thô không cần rời khỏi thiết bị.

ĐIỀU KIỆN QUAN TRỌNG

Thiết bị không chỉ suy luận mà còn phải **huấn luyện cục bộ -** mô hình phải đủ nhẹ để thiết bị tính toán và lưu bộ nhớ khi học.

1 Dữ liệu mỗi thiết bị khác nhau

Camera nhà máy A chủ yếu thấy sản phẩm A, nhà máy B thấy sản phẩm B → mô hình chung có thể khó hội tụ hơn.

2 Tốn tài nguyên trên thiết bị

Huấn luyện cần nhiều tính toán, bộ nhớ, năng lượng hơn inference. Một số thiết bị có thể mất mạng · chạy chậm · tạm không tham gia.

3 Vẫn có rủi ro riêng tư

Cập nhật mô hình vẫn có thể chứa thông tin về dữ liệu đã học. Khi cần riêng tư cao: Secure Aggregation (server chỉ thấy kết quả gộp) · Differential Privacy (thêm nhiễu).

FL giữ dữ liệu gốc tại thiết bị, nhưng còn ba bài toán: dữ liệu khác nhau · tài nguyên hạn chế · rủi ro từ cập nhật mô hình.

Sáu hướng nghiên cứu triển vọng

Vision Mamba / SSM

Làm mô hình nhìn ảnh với chi phí tăng chậm hơn khi ảnh lớn
Kết hợp SSM với tích chập, tối ưu lượng tử hóa và mở rộng sang detection/segmentation.

Lượng tử hóa 4-bit

Giảm model xuống 4 bit để tiết kiệm bộ nhớ
Có thể kết hợp 4-bit và 8/16-bit cho các phần khác nhau.

Huấn luyện ngay trên thiết bị

Thiết bị không chỉ suy luận mà còn có thể học thêm
Ví dụ camera hoặc robot cập nhật mô hình từ dữ liệu mới tại chỗ.
Mục tiêu: mô hình thích nghi với môi trường sử dụng thực tế.

Đưa mô hình lớn xuống thiết bị biên

Giữ khả năng của model lớn nhưng thay phần nặng bằng mạng nhỏ
Ví dụ với SAM: thay encoder lớn và dùng chưng cất tri thức.
Mục tiêu: giữ chức năng quan trọng nhưng giảm chi phí triển khai.

Đồng thiết kế HW–SW

Không thiết kế mạng trước rồi mới tìm chip để chạy
Kiến trúc mạng được chọn dựa trên những phép toán mà NPU/chip thực hiện tốt.

TinyML vision

Đưa mô hình thị giác xuống vi điều khiển rất nhỏ
Model ít tham số là một lợi thế, nhưng vẫn phải kiểm tra:
RAM · bộ nhớ chương trình · phép toán được hỗ trợ · độ trễ

Xu hướng chung: mô hình không chỉ cần chính xác hơn, mà còn phải nhẹ hơn, thích nghi hơn và phù hợp hơn với phần cứng thực tế.

Bốn kết luận chính

1

Chọn theo phần cứng trước

Biết chip đích và ngân sách p95, năng lượng TRƯỚC khi chọn mô hình. FLOPs không phải tốc độ.

2

Xếp chồng kỹ thuật

Hiệu quả có thể cộng dồn nhưng không có một tỷ lệ chung; báo cáo riêng storage, latency, RAM, năng lượng và accuracy.

3

Khả năng triển khai cũng là chất lượng

NMS-free, re-param, export sạch và mức suy giảm sau lượng tử hóa đều phải xác minh trên backend đích.

4

Theo dõi các hướng mới

SSM, số học 4-bit, federated learning, foundation model xuống biên → khoảng trống nghiên cứu.

Backbone và nén mô hình

BACKBONE

MobileNetV4 (Google, ECCV 2024) · iFormer (ICLR 2025) · RepViT (CVPR 2024) · SHViT (CVPR 2024) · MicroViT (ISCAS 2025) · TinyNeXt (ICCV 2025) · EfficientViM (CVPR 2025) · VMamba (NeurIPS 2024)

LƯỢNG TỬ HÓA

GPTQ · SmoothQuant · LSQ (learned step size) · GPLQ (2025, QAT cho ViT) · NVFP4 + Quantization-Aware Distillation (NVIDIA 2025–2026)

DISTILLATION · PRUNING · NAS

Hinton et al. (2015) · DeiT distillation token · CQKD (2025, nén hợp nhất) · Taylor pruning · hardware-aware NAS trong MobileNetV4

Gợi ý thứ tự đọc: MobileNetV4 → iFormer → RepViT → SHViT → EfficientViM; nén thì SmoothQuant → GPTQ → NVFP4.

Tác vụ, hệ thống và học liên kết

DETECTION

YOLO11 (Ultralytics docs, 2024) · YOLOv12 (2025, attention-centric) · YOLO26 (arXiv 2606.03748 — NMS-free, 2026) · DETR end-to-end matching

SEGMENTATION

MobileSAM · EdgeSAM · EfficientSAM (CVPR 2024) · EfficientViT-SAM (2024) · EfficientSAM3 (preprint 2025, arXiv 2511.15833)

FEDERATED LEARNING & HỆ THỐNG

Survey Federated Learning for Edge AI (2025, hơn 200 paper) · Energy-Efficient FL for Edge Real-Time Vision (arXiv 2508.01745) · tài liệu TensorRT, TFLite, CoreML, OpenVINO

CẢM ƠN

Thị giác máy tính hiệu năng cao trên thiết bị biên Trường hè Edge AI 2026 · SoICT — ĐHBK Hà Nội Samsung × NAVER · 10–14/8/2026

soict.hust.edu.vn/summer-school

ONE LOVE. ONE FUTURE.